Application Development Language v1.2

Preethi Pandian
ppandian@iastate.edu

Follow this and additional works at: https://lib.dr.iastate.edu/creativecomponents

Part of the Other Computer Engineering Commons

Recommended Citation
https://lib.dr.iastate.edu/creativecomponents/413
Application Development Language v1.2

by

Preethi Pandian

A Creative Component submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Simanta Mitra, Co-major Professor
Gurpur Prabhu, Co-major Professor
Ying Cai

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2019

Copyright © Preethi Pandian, 2019. All rights reserved.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Challenges and Objectives</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Organization</td>
<td>3</td>
</tr>
<tr>
<td>CHAPTER 2. MOTIVATION</td>
<td>4</td>
</tr>
<tr>
<td>CHAPTER 3. Application Development Language v1.2</td>
<td>5</td>
</tr>
<tr>
<td>3.1 General Idea</td>
<td>5</td>
</tr>
<tr>
<td>3.2 Architecture</td>
<td>6</td>
</tr>
<tr>
<td>3.3 Technology Description</td>
<td>6</td>
</tr>
<tr>
<td>3.3.1 Gradle/Maven</td>
<td>6</td>
</tr>
<tr>
<td>3.3.2 SpringBoot</td>
<td>7</td>
</tr>
<tr>
<td>3.3.3 Mustache</td>
<td>7</td>
</tr>
<tr>
<td>3.3.4 Websocket</td>
<td>8</td>
</tr>
<tr>
<td>3.4 Specification</td>
<td>8</td>
</tr>
<tr>
<td>3.5 Implementation</td>
<td>12</td>
</tr>
<tr>
<td>3.5.1 Parser</td>
<td>13</td>
</tr>
<tr>
<td>3.5.2 Server Generator</td>
<td>13</td>
</tr>
</tbody>
</table>
3.5.3 Client Generator .................................................. 16
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1</td>
<td>This table shows possible data types of attributes in model.</td>
<td>14</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>This table shows possible features in input spec and how they are automated in Controller</td>
<td>15</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>ADL v1.2 Automation in Controller</td>
<td>20</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Architecture Flow Diagram of ADL v1.2</td>
<td>6</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture Flow Diagram of generated code</td>
<td>7</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of Mustache</td>
<td>8</td>
</tr>
<tr>
<td>3.4</td>
<td>Spec file for Music Site</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Provided Input spec</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Generated Output Directory Structure</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Generated Server project structure</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>Auto generated Database Tables</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>Generated application properties file</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>A sample auto generated Controller Class</td>
<td>25</td>
</tr>
<tr>
<td>4.7</td>
<td>Generated Websocket project structure</td>
<td>25</td>
</tr>
<tr>
<td>4.8</td>
<td>Generated Client project structure</td>
<td>27</td>
</tr>
<tr>
<td>4.9</td>
<td>A sample generated login page</td>
<td>27</td>
</tr>
<tr>
<td>4.10</td>
<td>Registration page for sign up</td>
<td>27</td>
</tr>
<tr>
<td>4.11</td>
<td>Upload ArtistBio detail</td>
<td>29</td>
</tr>
<tr>
<td>4.12</td>
<td>Get all records for Users</td>
<td>29</td>
</tr>
<tr>
<td>4.13</td>
<td>Auto-generated Google Maps integration in UI</td>
<td>29</td>
</tr>
<tr>
<td>4.14</td>
<td>A sample input excel file for bulk upload</td>
<td>30</td>
</tr>
<tr>
<td>4.15</td>
<td>Data entry in MySQL db post upload</td>
<td>30</td>
</tr>
<tr>
<td>4.16</td>
<td>A sample auto-generated working chat feature</td>
<td>30</td>
</tr>
<tr>
<td>4.17</td>
<td>Input spec file for ABET</td>
<td>31</td>
</tr>
</tbody>
</table>
# NOMENCLATURE

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>App description Language</td>
</tr>
<tr>
<td>ADL v1.2</td>
<td>ADL version 1.2</td>
</tr>
<tr>
<td>CRUDL</td>
<td>Create, Read, Update, Delete, List</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain old java object</td>
</tr>
<tr>
<td>JSON</td>
<td>Javascript Object Notation</td>
</tr>
<tr>
<td>DDL</td>
<td>Data Definition Layer</td>
</tr>
<tr>
<td>PK</td>
<td>Primary Key</td>
</tr>
<tr>
<td>ORM</td>
<td>Object-relational Mapping</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer Architecture</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to Dr. Simanta Mitra for his substantial guidance and support throughout the course of this research. I would also like to extend my appreciation to my co-major professor Dr. Gurpur Prabhu and my committee member Dr. Ying Cai for their help and support.

In addition, I would also like to thank my friends, colleagues, the departmental faculty, and department staff for making this journey not only possible, but also a memorable one.

Last but not the least, I would like to extend my gratitude to my parents for their unbounded support and love.
ABSTRACT

ADL was an attempt at auto-generating web applications that included both client and server implementation. It served as a successful proof of concept and version 1.2 of ADL builds on top of its existing architecture and features. Many real world applications extensively use complex relationships between their database tables to store and fetch information. While ADL implemented one-to-many relationship, it is only natural to leverage the existing technique to implement many-to-many relationship. The other important aspect to focus is security: a login and registration page are default across all applications, authentication is a primary step towards adding security. Along with authentication, it’s also evident to protect server APIs from unauthorised access. ADL also did not provide different views for different users by capturing their permission levels. ADL v1.2 aims at closing in those gaps to further enhance the capabilities of this language. ADL v1.2 continues to function by parsing an input spec file and generates server-side code in Spring Boot and client-side in HTML and JavaScript, connected using XMLHttp Requests. Considerable changes are made to template files, including changes to code base, new tags are introduced in this version that supports more description for a given app. ADL v1.2 also supports all the features available in previous version.
CHAPTER 1. INTRODUCTION

When was the last time we implemented an HTTP Server from scratch? Probably not, very few people bother to implement an HTTP server because that task has been done. How about connecting server to an App by creating a basic REST API? Seems there’s a command line tool that does that now. This constant automation of software is why software tools get better every year. Automation of code generation is a very popular field that is garnering a lot of attention in recent times. The App Description Language (ADL) was a successful attempt at auto generating REST client with Spring Boot, and user Interface with HTML and JavaScript. The specification or language to auto generate code is simple enough to be written by anyone with little to no programming experience.

ADL v1.2 presented in this report adds more features to existing JSON language, in terms of security, complexity. This is language is useful for those, who are skilled to develop the whole application but have time constraints on rewriting the simple and mundane thousands of lines of code. Instead, the time and energy can be focused on building the more important parts of application i.e. logic and security aspects.

1.1 Background

A typical server-side REST application which follows MVC (Model-View-Controller) pattern, involves a main application class, plain java model objects for each entity or table in the database, controllers for exposing the endpoints, repositories corresponding to each model for Hibernate or any other ORM tool. Further, it requires settings and description of dependency management, and properties file to configure application level properties. The view is optional and can be omitted
as we will be developing the front-end UI separately.

The user interface/client-side is an interactive layer between the end-user and server-side code. jQuery is used to select and work on the HTML components. Together, Javascript with HTML and jQuery is definitely the most popular method of generating front-end application and tie it to the corresponding back-end. Due to increased demand of usability and manageability, integration of external APIs is pretty common in current applications. Google cloud platform provides easy and sophisticated APIs to integrate popular functionalities such as Maps, Drive, Calendar etc, and saves a huge amount of time by not rewriting the code for same functionalities.

1.2 Challenges and Objectives

The major challenge in developing app description language is to design the input spec file. The file should capture as much information as possible, at the same time minimizing the different input parameters and keeping it simple to understand and write a new one for users. The existing code generators like Swagger (9) primarily focuses on capturing the very relevant parameters, but it might be challenging for a non-technical person to design the same, although it supports multiple languages and can accommodate many features. Another challenge is to understand what the user wants and requires without explicitly having to mention it. For example, to get the list of all users based on a geographic location say city is to be captured by a simple understandable word like getUsersByCity and should be able to generate corresponding server-side and client-side.

Furthermore, since the logic and security aspects are something, which must be configured manually later on top of the generated code, it is important to keep the code modular, indented and easy to understand, so that developers can further work on it.

Another important aspect to consider is that fact that there is plethora of applications users might want to generate, and thus the language description should be able to understand the require-
ment and accommodated them. Hence our objective is to develop a simple description language, easy to parse and understand the users requirements and generate the code with high modularity and simplicity.

1.3 Organization

The next part of the report is organized as follows. Firstly, we talk about the motivation behind pursuing this project, and why it is intriguing and interesting to work. Next, we discuss about the core methodology, technology used and flow of information to automate the application development. Then, we experiment with the automation using two examples. One example is to verify and establish the automation achieved. Another one is to showcase its limitations and domain and scope. Lastly, we talk about the related works done in this domain, and possible future works.
CHAPTER 2. MOTIVATION

Initially, ADL was put to test by auto-generating a website for course management. The time to create an end to end functional app was certainly reduced, front-end was tweaked to include few additional features, back-end had few configuration changes to deploy at a remote server. All users for the app were identified, the app was hosted and made ready to use. It quickly came to notice that ADL lacked in security. ADL lacked to generate a basic login and registration page, a feature that’s commonly required across all application. Authentication is crucial and primary step towards security. In addition to login page, the auto generated server APIs should also be session covered, to additionally ensure no unauthenticated user has access to backend AP’s.

Secondly, the idea of authentication could be further extended to provide multiple views of same website for different kind of users. If the input spec file have provision to capture different users that are intended to use the auto-generated app along with their permission levels - permissions for CREATE, READ, UPDATE, DELETE, LIST - CRUDL, combined we can seamlessly create different UI views for appropriate users, enhancing capabilities of ADL.

Lastly, implementing many-to-many interaction between entities, many real world website and applications extensively use such relationships and by providing a facility to auto-generate such feature, would greatly increase the usability of ADL and enable create of complex web applications.
CHAPTER 3. Application Development Language v1.2

3.1 General Idea

The general idea behind ADL v1.2 is to add security and to capture different users and their permission levels to provide different views and accessibility to intended users of an application generated using ADL. This is achieved by creating an input specification file in JSON format. ADL parses the input file and internally creates a corresponding java object, which makes it convenient to access information of each tag without much overhead.

After the information is extracted, server-side Spring boot code is first generated with its appropriate pom.xml, application.properties file, all the specified mode entities which would correspond to tables in the database made possible by the use of Hibernate mapping provided by Spring boot configuration. Each entity repository and controller are also created by combing the base template and information provided in the spec file. The auto-generated code adheres to Spring boot guidelines, along with automatic DDL configuration, which would automatically generate the tables and relationships, when the generated application runs for the first time. A standard convention is followed for creating the API endpoints with corresponding GET and POST methods.

Client-side code is generated followed by the server-side. A default login page that links to a registration page is created, followed by a home page that serves a landing page after a successful sign in and additionally provides links to all entity pages created as per input file specification. Two javascript files are created - login.js referenced in index.html which is the login page and another js file - control.js referenced in every entity page. Two CSS style sheet pages are also created one: home.css applied to every UI component in entity pages, and, style.css applied to every UI component in index.html(login). HTML elements are mapped in Javascript by jQuery. Standard conventions are used to call Spring boot APIs to interact with the database. The entire code base is generated in folder location specified as an input argument for ADL.
3.2 Architecture

The figure depicts the generation of the server side and client-side code along with its integration.

3.3 Technology Description

3.3.1 Gradle/Maven

Gradle continues to be the choice of build tool for v1.2. Gradle supplements dependencies required for this project such as Apache StringUtils which is heavily relied on for String manipulation, a feature extensively used in this project. For the server code that is generated, Maven continues to be the choice of build tool with pom.xml as the file to specify maven configuration information and dependencies.
3.3.2 SpringBoot

ADL v1.2 server-side code generation is built on SpringBoot architecture which is the same as the previous version. SpringBoot MVC pattern and Hibernate feature along with its embedded Tomcat continues to be the choice of framework both for its ease of use and modularity.

![Architecture Flow Diagram of generated code](image)

Figure 3.2 Architecture Flow Diagram of generated code

3.3.3 Mustache

Mustache enables us to achieve the core feature of ADL, which is to automatically generate code for given different information. Mustache is a tag-based template language. It provides us with a powerful mechanism to replace tags to any specified value. Let us consider an example to better understand what Mustache does:

If the above-given String is considered a template, then by giving different values to variable enables us to generate different strings.
3.3.4 Websocket

The WebSocket feature of ADL v1.2 is untouched and retains the same functionality of the previous version. Care and sufficient tests have been done to ensure that this feature's behavior in unaltered in this version. Just as in (1), even in v1.2websocket is leveraged to create chat functionality. And the chat option continues to be a group chat, as in any user having access to the website auto-generated using ADL can start communicating with anyone else having access to the same website.

3.4 Specification

ADL v1.2 follows the same semantics as (1), except a few newly added tags, that follow the same convention of JSON datatype.

The spec file below demonstrates complete features ADL v1.2 currently supports.

1. **basePath:** If we want to our server-side APIs to precede a common path, we can specify that string in this tag. This is an optional argument, if not specified there will be no error in parsing or in code generation. Avoid special characters in the input. The basePath in the attached spec in Figure 3.4 is "/music".

```java
String text = "One, two, {{variable}}. Three sir!";
Template tmpl = Mustache.compiler().compile(text);
Map<String, String> data = new HashMap<String, String>();
data.put("variable", "five");
System.out.println(tmpl.execute(data));

// result: "One, two, five. Three sir!"
```

Figure 3.3 Example of Mustache
Figure 3.4 Spec file for Music Site
2. **title:** title tag captures the name of the root folder under which the entire client and server code gets generated. Also used as the package name for all server-side code. And finally, this name also appears in home.html - as a string describing the website. This is not an optional argument, failure to specify this detail will cause an error at run-time. Similarly, the title in the attached spec in Figure 3.4 is "Music-verse".

3. **description:** This information is used to provide value for the description tag in pom.xml that gets generated for server-side code. Again this is also an optional parameter, failure to specify this tag will not result in any parser or run-time errors. The description in the attached spec in Figure 3.4 is "A go to Database of your favorite Music".

4. **host:** Optional tag, sets the server URL for front end (in JavaScript files), if not provided then the default is set to localhost.

5. **port:** This is also an optional tag, used to set port value for server URL for front end, if not provided default is set to 8080.

6. **models:** It contains a list of various entities involved in the application, and each model contains a list of its attributes and corresponding data types. ADL v1.2 currently supports String, Integer, File and Timestamp data type. It also captures the relationship between 2 entities by mentioning the data type as either another entity name or by entity name followed by [] or by entity name followed by to represent one-to-one, one-to-many and many-to-many relationship respectively. In the attached spec in Figure 3.4, we have 5 models named User, Artist, Album, Song and ArtistBio. Also, "name" and "email" are the attributes of model "User", with both string data types. Similarly, other attributes can be deduced. The important thing to note here is the model "Artist". Out of it's 5 attributes, 2 are relationship mapping. Since one artist can have multiple Albums, and similarly Album can have multiple artist, therefore Artist has an attribute called "albumList" with type "Album", implying Many-to-Many relationship with Album. Similarly, Artist is related in One-to-One mapping with artistBio, assuming that one Artist can have one Bio. Also if you notice in model
Album, since one Album can have many songs, it has a One-to-Many mapping with Song, specified by attribute songList of type Song[]. [] is used to specify One-to-Many relationship. Moreover, an ArtistBio can have a picture, thus an attribute is mentioned as ”pic” with ”file” type. Any media type can be attached as file type. Similarly, an Album’s release date can be noted, thus the attribute called ”releaseDate” has the type as ”timestamp”, so as to store the time in database. Note that by mentioning the models, our intention is to create the tables with name same as model and the attributes as columns.

**item ui:** It is again an object containing various other important arguments, which are following:

(a) **platform:** Since there are multiple options to launch and expose the service such as web, mobile etc. Currently, ADL supports just web as the value of this attribute, but we can expand it to work for lot more such as Android, iOS etc.

(b) **loginuser:** Not an optional tag, this tag is used to specify among all models, which one is to be considered the entity for authentication, failure to specify this tag will result in run-time error in client-side generation. In Figure 3.4 it is User model that’s identified as login user.

(c) **permissions:** This also not an optional tag, this tag is best understood, if imagined as a table, with column headers being the type users specfied in User model, since permissions tag depends on user type attribute of login user, even type attribute of login user is mandatory. Row headers are all the models specific for database, each cell has a 6 character, where each character can take value ”1” for yes or ”0” for no, for Create, Read, Update, Delete, List activity respectively. For example in the attached spec in Figure 3.4, for model User and User type Admin, the value set is 11111, meaning Admin user has all permissions for CRUDL on User model.

(d) **features:** This is the most important part of the specification, as here we mention the services needed for the application, we are trying to build. The corresponding services
or features would then be exposed via a simple button in the corresponding models html page. We specify the features for each model individually. If there are no features corresponding to a model, the application will assume the corresponding entity as an inner or helper model. The queries corresponding to relationships are critical and must be mentioned carefully with a standard unified formatting technique. In the attached spec in Figure 3.4, the major features required are:

**getAll**: Get the list of all Artists currently listed in database

**getByName in Artist**: Get(Search) the Artist by the Artist name. Note that name is an attribute of Artist. The case is very important here. The first letter of attribute has to be capitalized along with the ’B’ in By.

**save in Artist**: This feature is to save a record in database, i.e. to add a row in Artist table.

**getAlbums in Artist**: Since Artists and Albums are related as Many-to-Many relationship, this feature is to get a list of all Albums in a given course. The same would hold for a One-to-Many relationship too.

**chat**: This feature is to enable chat in the concerned screen developed via Websockets.

(e) **integrations**: In this part, we provide the 3rd party integration we might need. It is again, has to be mentioned corresponding to individual model. If no integration is needed for any model, then this part can be ignored. Currently ADL supports just Google Maps integration. But maps integration provides a solid evidence that most of the other integrations such as Calendar, Drive etc can be integrated similarly. The ”maps” integration requirement is mentioned in the input spec in Figure 3.4 under integration and corresponding model.

### 3.5 Implementation

ADL v1.2 has in total 9 classes with ADLApplication being the main class. It takes 2 arguments, one, a input specification JSON file, two, a target directory where the auto generated files
get stored. When main() function gets invoked it requests for these 2 arguments and calls the checkParser() function of parser class.

checkParser() calls on ObjectMapper to map the input JSON to an internal Java object, making it convenient to retrieve necessary information. After objectMapper control is passed to Server Generator and then Client Generator class respectively to handle automation of code.

Constants class keep note of all necessary information on template files, like their location, no. of lines. Utilities class provides us with functions to read or write from files. And StringGenerator a class newly introduced in this version handles all String manipulation in a single place to increase readability of code. ADL v1.2 is also a Gradle project supporting with its exhaustive libraries for String manipulation, File handling and JSON processing.

### 3.5.1 Parser

Parser class has only one function checkParser() and only 2 responsibility, first it converts the input spec file into a Java object using objectMapper and then calls ServerGenerator and Client Generator class. For objectMapper to work, we need a corresponding Java class for it to map to. And for that purpose, we have 2 java classes written, input.java and UI.java, both model the tags of input spec file. ObjectMapper’s readValue() function would parse through the input spec file and populate corresponding attributes in input and UI class.

### 3.5.2 Server Generator

After the input spec file is parsed, the input java object is passed to server generator. We have three important functions here but before, all template files are copied to the target directory. The template files are of 2 types, one which require no modification, like MainServerApplication, pom.xml and application properties. We then use Mustache to load the standard files as template and replace the customizable field using the values obtained from the input spec. The second
category is the files that requires heavy customization, which are Controller, Model and Repository files. Since, these files are to be created for each model present in the spec, we first create a copy with the appropriate naming convention for each model, and then we work on all 3 files corresponding to each model, one by one. The three functions overall working is described below. Additionally we also identify the login user from loginUser tag from ui and store it in a global variable, the login user model will be a special entity which will have additional API for /login and /registration. We also create a map to store all model pairs that have Many-to-Many or one to Many mapping.

3.5.2.1 Models

Name of the model class is the same model name, provided in spec. We annotate each model with @Entity and @Table. Then each attribute for a model is considered and we add a declaration of the same with @Column spring annotation, along with getters and setters. For file type attribute, we save it as String type, as we save the file in a server location, and then just store the absolute path of the file in database. For timestamp, we use Java Util Date format. The time is stored as current system time in local time format in the database. Table 3.1 presents various data types supported by input spec and how they are automated in ADL.

<table>
<thead>
<tr>
<th>Data type of attribute parsed from spec, for a model M</th>
<th>Automation in Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>String/file</td>
<td>String type, Only @Column annotation</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer type, Only @Column annotation</td>
</tr>
<tr>
<td>X, where X is another model</td>
<td>X type, @OneToOne mapping along with @JoinColumn annotation</td>
</tr>
<tr>
<td>X[], where X is another model</td>
<td>List&lt;X&gt; type, @OneToMany mapping along with @JoinColumn annotation</td>
</tr>
<tr>
<td>X{}, where 'X' is another model</td>
<td>List&lt;X&gt; type, @ManyToMany mapping along with @JoinColumn annotation</td>
</tr>
<tr>
<td>timestamp/date</td>
<td>Date type, @Column annotation</td>
</tr>
</tbody>
</table>

Table 3.1  This table shows possible data types of attributes in model
3.5.2.2 Repository

Auto-generation of Repository is unchanged from previous version. ADL v1.2 follows the same naming convention, model name appended by string Repository. Each Repository is an interface and extends Spring JPARepository for hibernate, which provides function to interact with Database like find All, find by Id(pk) and save. If there any other feature specified in UI feature tag apart from these queries, then that function gets added in Repository.

3.5.2.3 Controller

<table>
<thead>
<tr>
<th>Feature (for model 'm')</th>
<th>Method</th>
<th>Endpoint Created</th>
<th>Automation in Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAll</td>
<td>GET</td>
<td>/m/all</td>
<td>A method and return the results of findAll method from Repository</td>
</tr>
<tr>
<td>getByX, where X is an attribute of m</td>
<td>GET</td>
<td>/m?X=a, a is query variable</td>
<td>A method and return the results of findByX method from Repository</td>
</tr>
<tr>
<td>getXs, where X is another model having a relationship with m</td>
<td>GET</td>
<td>/m/a/XList, a is query variable</td>
<td>A method to first get the row in m for id a, then return the list of X present in the object.</td>
</tr>
<tr>
<td>save</td>
<td>POST</td>
<td>/m</td>
<td>A method and return the results of save method from Repository</td>
</tr>
<tr>
<td>bulkUpload</td>
<td>POST</td>
<td>/mBulkUpload</td>
<td>A method that accepts an excel sheet with first row as header, and data in subsequent rows, along with another list of files as Multipart[], each file (eg: image or pdf) corresponds to each row in the same order</td>
</tr>
<tr>
<td>chat</td>
<td>Websocket</td>
<td>/websocket/x, x is username</td>
<td>WebsocketConfig and WebsocketServer to handle messages</td>
</tr>
</tbody>
</table>

Table 3.2 This table shows possible features in input spec and how they are automated in Controller

Since the focus of this version is on Security and we want to restrict access to server APIs by using sessions. The method signature for all function API is completely changed. For previous version Table 3.2 consists of various possible options for feature generation from ADL, it also shows
how the code is generated for the same. For the current version Table 3.3, all API take in additional argument HTTP Session to validate if the user has actually logged in, if an unauthorized user tries to access the API, then session will be NULL for such user and they would receive a response HTTP.404. This way we are protecting every endpoint and to facilitate sending different return types, for example, a response HTTP.404 for unauthorized user and say list of all entries for an authorized user, we will be using a wrapper class, ResponseEntity, enabling us to send both. Additionally, version 1.2 also supports Update and Delete API.

3.5.2.4 WebSockets

WebSockets feature of ADL v1.2 is same as ADL, no modification done to its functionality and its been tested in latest version to see its behavior in unaltered and no bugs are induced. WebSocket component is built only when UI-feature tag has value ”chat” for at least one of its screens. The backend gets built with standard WebSocketConfig and websocketServer classes along with Main Application, this is generated separately for simplicity and the design is open to additional modification for more advance chat features. But currently, it continues to build a group chat feature with methods, onOpen, onMessage and onClose. This is entirely built by mustache and template files with minimal customization.

3.5.3 Client Generator

Client-side is a web application, developed in HTML, CSS and JavaScript. First, all files related to login and registration gets generated, which now becomes the starting point of application. The design of front end is still restrictve, every style is preconceived and generated in one specific way. But it’s open to customization after generation. Login page has link to registration page, on successful sign up, control passes to login page again. After entering the right credentials, 2 things happen, first the session information for the user gets stored, second, we response from server, the type of user logged-in, based on this info, all the upcoming pages and permissions get set. After
login page, comes home.html which is landing page after successful login and has hyperlinks to all entity pages. CSS files are also fixed for model or entity pages.

### 3.5.3.1 Login Files

This functionality is an addition in ADL v1.2. As discussed in section 3.5.2.1, we have already identified the login entity and it’s variable. A login HTML file is gets generated by copying the template login.html file and login.js file gets generated by help of Mustache. login.js file has appropriate call to server API’S for login validation and storing Session. A registration page is also generated here, we parse through each attribute of this entity and generate an HTML input form. A save button is also generated and that is linked to appropriate back-end API for registration.

### 3.5.3.2 Model HTML Files

This function is also modified under v1.2, previously we would iterate through all models specified under ui components in spec file to create HTML for each model. But in this version in addition to iterating over each model, we also create versions of same model for different user types, based on their values set under permission tag, for example from Figure 3.4 under permission tag say for model Artist for user type Publisher the value provided is 01001, here 01001 stands for its CRUDL, so a publisher user type has no Create permission, so no save feature, no delete or update feature, only get by attribute and get ALL which corresponds to Read and List is permissible, therefore the version of page created for this particular entity will have only getBy and getAll components. Each model file creation follows the same naming convention user type appended by model name. While iterating each model, we also create a map for model pair that have Many-to-Many or one-to-many relationship.

The remaining process of how the file gets generated using mustache is same, the template file is copied to appropriate location in output directory, we add hyper-links to all other html pages as
buttons, including index to interlink the whole project. Apart from hyper-link buttons, we add the
div form element for each type of "post" feature present in the input spec, i.e. save and bulkUpload
as mentioned in Table-3.2. In case of save, we add the input HTML element for each attribute
present in the model, and in case of bulkUpload, we add input element for excel file and collection
of individual elements, in accordance with the server-side API. These div elements are initially
hidden and we expose them through buttons in the model page, and each feature corresponds to a
button. For GET features, as per Table-3.2, upon clicking the corresponding data is presented in	tabular view. For POST features, it’s the corresponding form, which is loaded.
Further, we also search the integrations component in the spec to look for whether the same model
requires any 3rd party integrations, if so, we just add the div element accordingly. The config and
Google APIs are already integrated in the template.

3.5.3.3 JavaScript Files

Even in version v1.2 a single JS file is generated with methods corresponding to each operation
in all the model pages. Similar to other pages, we load it through a template file. The major task
of javascript file is to make Http calls to server side, thus we already have the basic functionality of
a GET and a POST http call through XmlHttpRequest implemented and added, additionally we
also have PUT and DELETE http requests for Update and delete actions. Now, as we consider
each model present in feature at a time, we go through all operations needed with the server-side
along with their permissions. Each button corresponding to a GET element in html is mapped
to make a Http GET request to the corresponding API and once the result are fetched, the data
is converted to tabular form by adding each JSON object from the JSON array to the table as a row.

Further, for each button corresponding to a POST element in html is first mapped to launch
the corresponding div form, and then once the data is filled out by user, input JSON request body
is compiled with the same data, including the files, if provided. Then the submit button in the form
is mapped to make a Http POST request to the corresponding API. As mentioned in section 3.5.3.2 we have map of all model pairs for Many-to-Many and one to many. Whenever we encounter post option on such models, for many to one, we display a list of all entries of other model with radio button, to save the join relationship between 2 models, for Many-to-Many model pair, similarly for post option instead of radio button, check boxes are displayed. The success and failure of the request is then notified. With the various mappings happening between model file and javascript file and then to the server-side APIs, it is quite evident that the naming convention has to be uniform, unique and properly defined for each type of operations, which we try to do by using the same names for operation as provided by user in feature to keep uniformity between client-side and server-side.

### 3.5.3.4 Web sockets

As discussed in Sever generation of webSockets, even client side feature of webSockets retains same behavior as ADL, websocket javascript file is only created if there is a requirement of chat feature in at least one model. This javascript file simply makes a Websocket call to the respective server-side method, which implements onOpen, onClose and onMessage methods. Further, the chat box in the concerned model is added, which is implemented via a simple html table. With the help of jquery, the chat table is made dynamic to enable live chat rendering.
<table>
<thead>
<tr>
<th>Feature (for model 'm')</th>
<th>Method</th>
<th>Endpoint created</th>
<th>Automation in Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAll</td>
<td>GET</td>
<td>/m/all</td>
<td>A method and return the results of findAll method from Repository if session exists else return HTTP.404</td>
</tr>
<tr>
<td>getByX, where X is an attribute</td>
<td>GET</td>
<td>/m?X=a, a is query variable</td>
<td>A method and return the results of findByX method from Repository if session exists else return HTTP.404</td>
</tr>
<tr>
<td>getXs, where X is another model having a relationship with m</td>
<td>GET</td>
<td>/m/a/XList, a is a query variable</td>
<td>A method to first get the row in m for id a, then return the list of X present in object only if session exists else return HTTP.404</td>
</tr>
<tr>
<td>save</td>
<td>POST</td>
<td>/ml</td>
<td>A method and return the results of save method from Repository, i.e status HTTP.200 if session exists else return HTTP.404</td>
</tr>
<tr>
<td>bulkUpload</td>
<td>POST</td>
<td>/mBulkUpload</td>
<td>A method that accepts if session exists an excel sheet with first row as header, and data in subsequent rows, along with another list of files as Multipart[], each file (e.g., image or pdf) corresponding to each row in same order else return HTTP.404</td>
</tr>
<tr>
<td>chat</td>
<td>Websocket</td>
<td>/websocket/x, x is username</td>
<td>WebSocketConfig and WebsocketServer to handle messages</td>
</tr>
<tr>
<td>update</td>
<td>Update</td>
<td>/m/update</td>
<td>A method and return if session not null status HTTP.200 on successful updation else return HTTP.404</td>
</tr>
<tr>
<td>delete</td>
<td>Delete</td>
<td>/m/delete</td>
<td>A method and return if session not null status HTTP.200 on successful deletion else return HTTP.404</td>
</tr>
</tbody>
</table>

Table 3.3 ADL v1.2 Automation in Controller
CHAPTER 4. EXPERIMENTAL EVALUATION

4.1 Experiment Settings

All the experiments were performed in a standard IDE for Java with Java 1.8 virtual environment. The program seeks one input file and one output directory location with absolute path. The generated code has been tested by running the server-side code in a Windows 10 PC, with MySql server set-up in the same machine, and the client-side UI is tested in Google Chrome and Mozilla Firefox browsers.

4.2 Experiment 1

Scenario: An application for managing Albums, Artists and Songs published.
So, in detail, there would be User, Artist, Album, Songs and ArtistBio. An artist can have released many albums. And each Album could be released by multiple artists and each Album had many songs. Each Artist has a Bio page that describes him/her. And the application would have user who could be Publisher, Admin who maintains the site and normal user. Do note that we are also incorporating features that were supported in previous version, to demonstrate that current version had backward compatibility.

Input JSON Spec:
Figure 4.1 provides a comprehensive input spec file which captures almost all features; ADL v1.2 can automate at this moment.

Parsing the input:
As described in Section 3.4, the spec contains 5 models which are User, Artist, Album, Song and ArtistBio, with respective attributes. Artist has a One-to-One relationship with ArtistBio, Many-to-Many with Album and Album has one-to-Many relationship with Song. Also, each Artist can
Figure 4.1  Provided Input spec
have a corresponding file(image/pdf/doc etc).

Further, w.r.t features, in addition to simple getAll and save feature, we need bulkUpload and getAlbumss for a given artist in Artist; get Song information by name in Album; and chat feature in Artist.

![List of files]

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>7/9/2019 2:40 PM</td>
<td>File folder</td>
</tr>
<tr>
<td>server</td>
<td>7/9/2019 12:09 PM</td>
<td>File folder</td>
</tr>
<tr>
<td>websocket</td>
<td>7/9/2019 1:25 PM</td>
<td>File folder</td>
</tr>
</tbody>
</table>

Figure 4.2 Generated Output Directory Structure

**Server-side:**

Figure 4.2 shows the autogenerated folder structure. It is generated in the mentioned output directory, with separate server, client and websocket directory.

Now, Figure 4.3 shows the project structure generated for the server-side code. We verify that pom.xml, application.properties and ServerApplication is generated in appropriate directory, and similarly Controllers, Models, and Repositories are generated for each model mentioned in spec.

Further, spring related config is verified in application.properties file, which is attached in Figure 4.5. Default properties are set for server port, database driver(MySql), and DDL. The database url is set for default local, which can be modified manually later in case of production or any other instance of mysql db.
Music-verse_server [boot]

src/main/java

- com.example
  - Album.java
  - AlbumController.java
  - AlbumRepository.java
  - Artist.java
  - ArtistBio.java
  - ArtistBioController.java
  - ArtistBioRepository.java
  - ArtistController.java
  - ArtistRepository.java
  - ServerApplication.java
  - Song.java
  - SongController.java
  - SongRepository.java
  - User.java
  - UserController.java
  - UserRepository.java

- src/main/resources
  - JRE System Library [javaSE-1.8]
  - Maven Dependencies
  - src
  - target
  - pom.xml

Figure 4.3 Generated Server project structure

Figure 4.4 Auto generated Database Tables

```properties
server.port=8080
server.servlet.context-path=/music

spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/music
spring.datasource.username=root
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update
```

Figure 4.5 Generated application properties file
public class AlbumController {

    @Autowired
    private AlbumRepository albumRepository;

    @RequestMapping(method=RequestMethod.GET, path="/album")
    public ResponseEntity getAlbumByName(@RequestParam("name") String name, HttpSession session) {
        String activeUser = (String) session.getAttribute("user");
        if(activeUser==null)
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else
            return new ResponseEntity<>(albumRepository.findByName(name),HttpStatus.OK);
    }

    @RequestMapping(method=RequestMethod.GET, path="/album/{name}/artistlist")
    public ResponseEntity getArtistListForAlbumName(@PathVariable("name") String name,HttpSession session) {
        String activeUser = (String) session.getAttribute("user");
        if(activeUser==null)
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else
            return new ResponseEntity<>((albumRepository.findById(name).get().getArtistList,HttpStatus.OK));
    }

    public ResponseEntity getSongListForAlbumName(@PathVariable("name") String name,HttpSession session) {
        String activeUser = (String) session.getAttribute("user");
        if(activeUser==null)
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        else
            return new ResponseEntity<>((albumRepository.findById(name).get().getSongList,HttpStatus.OK));
    }
}

Figure 4.6  A sample auto generated Controller Class

Figure 4.7  Generated Websocket project structure
Figure 4.4 shows the automatically generated tables in the database, upon running the Server-Application. The columns and types are verified and they align with the required data types.

Now, if we consider a generated Controller class say AlbumController in Figure 4.6, generated based on the features corresponding to Album in input spec in Figure 4.1. Notice how the method signature in this version’s controller are different from ADL.

Figure 4.7 shows the generated Websocket project, as the input spec requires a chat feature in Comment page. The project is a simple Websocket project in Spring framework.

**Client-side:**

Figure 4.8 shows the layout of client-side project, with html, css and javascript files. home.css file contains all the style elements of the project. For changes in UI/UX pattern, this css file can be further modified. The basic functionalities for main application is contained in generated control.js, which calls the server-side APIs for integration. The websocket related integration is done in websocket.js. Lastly, features corresponding to each model is implemented in remaining html files, with index.html being the login page of the application.

Notice how type is generated as radio button for selecting the type of users in 4.10. Once registration is successful control is redirected to login page Figure 4.9, where the user is prompted to enter their credentials once, the credentials are verified by backend, server returns the type of user, and based on the user type, the appropriate home page is called, whose view will be different from rest of the users. Even in each model HTML page, the actions for different users are restricted by their CRUDL values.

Now, if we consider the feature of saving an ArtistBio, Figure 4.11 shows a basic form element to capture the attributes of ArtistBio model, as mentioned in the corresponding model. Since Image
Figure 4.8  Generated Client project structure

Figure 4.9  A sample generated login page

Figure 4.10  Registration page for sign up
attribute is mentioned as file type, the auto-generated form seeks a file to be uploaded. Currently, the relationship saving is not supported in ADL v1.2, which is listed as a future work.

Similarly, getAll feature is verified by Figure 4.12. This table is populated by calling the GET API generated in server-side application for Get all Users from User table in database. The tables are formatted as Datatables (12), which provides a nice template for tables with in-built sorting, searching and pagination feature.

Another important feature is the bulk upload feature, which is again mentioned as a requirement for artist model in the input spec. This form for bulk upload shown in Figure 4.14 which seeks an excel file, each row represents a data entry in Team table in database, first row is reserved for header, the order of which needs to match with the attribute order.

Next, Figure 4.13 verifies the auto-integration of Google Maps API, mentioned under integration component in the input spec.

Upon successfully submitting the bulkUpload form, we query the database to verify the save, shown in Figure 4.15.

Last, but not least, we verify the working of chat feature via websocket, the chat button appears on Comment page, as we mentioned in the input spec. The chat is integrated via websocket.js file, and multi-users can chat together in the chat box.

### 4.3 Experiment 2

**Scenario:** ABET - Develop an application to display individual course information or a report of each course.

**Outcome:** Clearly the backend generated by ADL v1.2 is very robust and has capabilities untapped by UI. The UI generated is per model, but the requirement is to generate a report, that takes in field across multiple models. The server has APIs to support this functionality, but it is restricted by the template design of UI, which relies heavily on table format.
Figure 4.11  Upload ArtistBio detail

Figure 4.12  Get all records for Users

Figure 4.13  Auto-generated Google Maps integration in UI
Figure 4.14  A sample input excel file for bulk upload

Figure 4.15  Data entry in MySQL db post upload

Figure 4.16  A sample auto-generated working chat feature
Figure 4.17 Input spec file for ABET

```json
{
    "title": "ABET",
    "basePath": "/abet",
    "description": "Iowa State University - ABET",
    "models": {
        "Course": {
            "number": "integer",
            "name": "string",
            "creditAndHours": "string",
            "courseInfo": "string",
            "preReq": "string",
            "EorSorSE": "string",
            "outcomes": "Outcomes"
        },
        "Outcomes": {
            "number": "integer",
            "one": "string",
            "two": "string",
            "three": "string",
            "four": "string",
            "five": "string",
            "six": "string"
        }
    },
    "ui": {
        "platform": "web",
        "features": {
            "Course": "getAll, getName, save, bulkUpload, getForOutcomes"
        }
    }
}
```
4.4 Measure of Success

We define following measures of success to evaluate the performance of ADL.

4.4.1 Functionality

As we see, ADL v1.2 provides additional functionality like Login, Registration, authentication in server side, stores sessions to validate users, Many-to-Many relationship is also implemented. Definitely there is a huge scope of further addition to the features, which we talk about in Chapter 6. Moreover, the generated code provides an advanced starting point for developers, who might want to add more personalized features.

4.4.2 Efficiency

Understanding and writing the input spec is easy and definitely will get more faster with subsequent uses. The generation of whole desired application upon passing a correct input json takes few seconds, which is way more faster than designing and coding the whole application manually, which might take more than a month.

4.4.3 Maintainability

Both the App Description Language application and generated code is very easy to maintain due to the choice of technology used, standardized naming convention and code quality.
CHAPTER 5. RELATED WORK

Automatic Code generation is a relatively new topic, but there has been considerable amount of work and effort that has been put into it. Swagger Codegen (9) is the most popular code generators available. This project is inspired by Swagger codegen’s implementation idea. Swagger Codegen uses Mustache to create files from the existing template. It is a very general purpose automation tool, that supports more than 50 languages and frameworks, and the user has the option to choose from them to generate the server-side. Swagger doesn’t provide the client-side automation, but Stirewalt and Rugaber et. al (3) discusses a mechanism to automate the client-side using HTML and Javascript through a tags based specification.

Moreover, Swagger doesn’t provide any functionality in the generated code, it lays out the foundation and basic boiler plate code, albeit it can generate the layout for a really complex scenario. Once it generates the code, the developer has to further work on understanding the codebase and then further work on it to add the functionalities like use Hibernate to fetch records from the database.

Another area, which I believe this project focuses on is the fact that ADL can be used by a person with little to no programming experience, unless the requirement is highly complex. Whereas Swagger demands the user to have pre-knowledge about all possible options of development. It also requires to manually set endpoint paths, http method type, authorization tokens, and lot more. Although, these provide a great range of customizations, but sometimes it might get overwhelming for a person.
CHAPTER 6. CONCLUSION AND FUTURE WORK

In this project, we added more capabilities to ADL. We have improvised on security, provided default login and registration page, added sessions to protect every server API. Enhanced interaction with database by implementing many to many relationships. The front end accommodates more UI components than before. We also achieved different viewing and accessibility of application by considering permission values for CRUDL. Having added the above functionalities, we have made auto generation of application to include more complex features, greatly reducing overhead of a naive developer and leaving only business logic to incorporate for advance developers.

For future work, one immediate area of focus could be auto-generating front end using a framework like React or Angular to improve maintainability and usability of client-side code. Once the client-side generation gets ported to a framework architecture we can further explore options for providing user the facility to design their own look and feel for front end, currently user of ADL is restricted in designing front end. Additionally, we could also venture the option of scrapping the whole option of writing an input spec JSON, because even that involves some syntactical knowledge of how a JSON object should be written, we could possible provide further abstraction, by generating JSON file internally, by making ADL user interact with a UI. Lastly, options to automate Plugin architecture could also be considered, so that the generated app can directly interface with various IDEs or external APIs.
REFERENCES


[5] ”Mustache GitHub Documentation” https://github.com/samskivert/jmustache,


[8] ”Maven Documentation” https://www.maven.co/,

[10] "Data Definition Layer Documentation"  


[12] "Datatables" https://datatables.net/

[13] "Websockets"  
https://git.linux.iastate.edu/vamsi/WebSockets/tree/master,  
https://en.wikipedia.org/wiki/WebSocket